RHD

STM32 Firmware Framework

Version 1.0

6 December 2023

Features

- Open-source STM32 firmware written in C to stream real-time data from Intan RHD2132 or RHD2216 electrophysiology amplifier chips.
- Optimized SPI communication between STM32 MCU and Intan RHD chip.
- Interrupt-based code using DMA (direct memory access) allows for sampling rates up to 20 kSamples/s per channel for 32 channels.
- Both HAL and LL libraries included.
- STM32U5 family supported.

Applications

- Rapid prototyping of Intan Technologies RHD amplifier-based products
- Starting point for the development of custom interfaces to RHD2132 or RHD2216 chips

Description

To facilitate the development of electrophysiology recording systems using the RHD series of microchips, Intan Technologies provides the following open-source STM32 firmware framework for developers. The framework consists of C code written for the commercially-available STM32U5 microcontroller family from STMicroelectronics. The example code streams multi-channel data from an Intan RHD2132 chip at sample rates up to 20 kSamples/s per channel using timers, interrupts, and DMA to maintain high throughput while using only a small fraction of the MCU capacity.

All Intan example code described in this document was developed on the inexpensive STMicroelectronics NUCLEO-U5A5ZJ-Q development board, which is available from many electronics distributors including DigiKey, Mouser, and Newark. Intan Technologies does not sell any MCU development boards.

Why Is This Code Specific to the STM32U5?

There are several families of STM32 microcontrollers with a very wide range of specifications suitable for different applications. The example code we provide here for the STM32U5 series is not an indication that only the STM32U5 chips will be the best fit for all projects using Intan chips. However, there are some specific advantages that the U5 family has over other STM32 series that make it a reasonable starting point for working with Intan chips.

The U5 series was launched in 2021 and is currently the most cutting-edge evolution of the well-established L series. It targets ultra-low power applications while still providing a maximum CPU clock rate of 160 MHz and robust peripheral support. High-speed processing and SPI data transmission / reception are the most important features for achieving high sample rate / high channel count communication with Intan chips, and while we have not quite been able to reach the maximum data rate the Intan RHD2132 chip is physically capable of supporting (30 kS/s for 32 amplifier channels + 3 auxiliary commands), we have gotten quite close (20 kS/s) with relative ease, using both the HAL (Hardware Abstraction Layer) and LL (Low Layer) drivers. The U5 series is should be sufficient for basic data acquisition from a single RHD2132 or RHD2216 chip and transmission of that data over a USART interface. Even those applications requiring a high sample rate will likely be able to achieve this by optimizing the interface to omit unnecessary features.

The most important features of the STM32U5 used in the Intan RHD firmware framework are **SPI** (Serial Peripheral Interface), **timer-generated interrupts** to achieve a reliable sample rate, and **DMA** (Direct Memory Access) to allow multi-word SPI transactions to occur without requiring direct processor intervention. Most applications will also require some way to transmit acquired data somewhere or save it to memory, so peripherals for interacting with USART, Ethernet, wireless systems, or SD cards, etc. will probably be useful.

SPI Communication Requirements

A critical signal in the Intan SPI communication protocol is $\overline{\text{CS}}$ (active-low chip select, called NSS in the STM32) rising high, remaining high for at least 154 ns, and then falling low between each 16-bit word. Unfortunately, the popular **STM32F4** series SPI bus does not appear to have an easy way to achieve this behavior. NSS is indeed driven low during each 16-bit word, but **NSS is not toggled high between words**, so the RHD chip does not receive the clear NSS/$\overline{\text{CS}}$ high signal indicating the end of a 16-bit word.

This NSS/$\overline{\text{CS}}$ pulse between every SPI word is required for the RHD chip to operate correctly, so for the STM32F4 chips we are forced to decouple NSS from the SPI peripheral and instead use a GPIO pin for $\overline{\text{CS}}$. Unfortunately, this requires direct processor intervention between every 16-bit word to write $\overline{\text{CS}}$ high, wait, and then write $\overline{\text{CS}}$ low again. This does allow the RHD chip to communicate properly, but it wastes CPU clock cycles, and prohibits the use of DMA for bulk data transfers. While this approach may be feasible for relatively low sample rates (5 kS/s or lower), it is inefficient and limits the communication between the MCU and the Intan chip. This manual control of NSS/$\overline{\text{CS}}$ is necessary for the F4 family and other STM32 families with similar SPI buses. (In theory it is possible to use a precisely-set timer tied to $\overline{\text{CS}}$ that is somehow synchronized with the SPI bus to automate $\overline{\text{CS}}$ toggling, but this complicates the SPI communication beyond the scope of entry-level demonstration code.)

The SPI bus implementation details can differ quite significantly between microcontroller family and manufacturer, so we strongly encourage users research their proposed MCU's SPI implementation to ensure this or similar shortcomings do not hinder or complicate data transfer with the RHD chip. We have verified that the **STM32U5** and **STM32H7** series have SPI buses that can easily be used with NSS automatically pulsing high between 16-bit words, and these families also work nicely with DMA for large data transfers.

Overview of Program Flow

This Intan STM32 example code was developed using STM32CubeIDE, and uses HAL or LL drivers (this can easily be changed by the user) to configure and control various peripherals.

The code first configures and initializes the MCU peripherals with various auto-generated functions, which the STM32CubeIDE environment creates based on various settings in the .ioc file. Parameters for configuring the RHD2132 chip are then used to determine values for each of the RHD registers, and these are written to the chip using WRITE commands over the SPI bus. A list of 32 CONVERT commands (one for each channel of an RHD2132 chip) and a list of three auxiliary commands (one of which continually re-writes the above-determined register values to the RHD chip) are created and stored in memory as *command_sequence_MOSI* for later use. The green LED is illuminated to indicate when data acquisition starts, and a timer interrupt is enabled. By default, this interrupt should trigger every 8000 clock cycles at 160 MHz, which is 20 kHz. We will refer to this timer period as the sample period. The program this enters a data acquisition loop.

This data acquisition loop will only break when the *loop_escape()* function returns 1, which will not occur until the number of acquired samples reaches 20000, meaning one second of data has been acquired. Until that time, this main loop will repeatedly write a dedicated pin *Main_Monitor_Pin* high, which can be used to monitor when this main loop is processing. (Other functions that trigger due to interrupts will keep this pin low for the duration of their execution). Because timer interrupts had just been enabled, this loop will consistently pause every sample period to execute *TIM_interrupt_routine()*.

The *TIM_interrupt_routine()* function executes once per sample period and begins sending a sequence of SPI commands. By default, this function sends 35 commands: 32 CONVERT commands + 3 auxiliary commands, where each command is a 16-bit SPI word. In addition to beginning the SPI transfer, this routine also writes the *Main_Monitor_Pin* low. (The main loop will write it high once it begins executing again.) The function also writes a dedicated *Interrupt_Monitor_Pin* high only for the duration of the function, and does some error checking. The SPI transfer uses DMA to iterate through the full 35-command list and this routine only begins the transfer, so by the time the routine finishes the 35-command sequence will have only just begun.

An important detail of *TIM_interrupt_routine()* is that it checks to make sure that the previous SPI sequence is complete; if the variable *command_transfer_state* (discussed below) is still *TRANSFER_WAIT* from the previous sequence, then the critical *ITClip* error has occurred. This error, as well as methods to avoid it, are discussed in more detail later. Briefly, this error indicates that the sample period is shorter than the time required for each SPI sequence, so every sample period the next sequence is being triggered before the previous one finishes. This can be solved by extending the sample period (i.e., using a lower sample rate) or speeding up each SPI sequence (e.g., sample fewer channels, use fewer AUX commands, or speed up the SPI transfer itself, if possible).

Eventually, one complete SPI transfer sequence will end. The exact way this is detected varies slightly based on whether LL or HAL drivers are used, but in both cases an interrupt triggers the function *SPI_TxRxCpltCallback()*. The *write_data_to_memory()* function is then executed, which is a user-changeable function that by default writes the result of a single CONVERT command (i.e., a sample from a single channel) to memory. Then, *transmit_data_realtime()* is executed, which is left blank in our example code, but is where the user could choose to implement some data transfer (for example using USART) to send out the MISO results from the now-finished command sequence before the next sample period overwrites the results. The variable *command_transfer_state* is changed; it is set to *TRANSFER_WAIT* once the transfer begins, *TRANSFER_COMPLETE* once the transfer finishes, and *TRANSFER_ERROR* if some error was detected.

The main loop will continue to run, pausing for an interrupt every sample period, until *loop_escape()* returns 1. If the user changes *loop_escape()* to always return 0 then this will never occur, so the program will simply continue looping with occasional interrupts. This would be suitable behavior for long-term data acquisition that is not directly saved to memory, but transmitted elsewhere in real time every sample period. Our simple example code is set to escape the loop after one second of data acquisition and disable further timer interrupts.

Next, the user-changeable function *transmit_data_offline()* is called, which by default sends the 20000 accumulated samples from a chosen channel out via USART in a single bulk transfer. (Implementing real-time transfers would require many smaller transfers to be made continually.) Note that the on-chip memory limitations of the STM32 will not allow for long recordings, especially at high sample rates and channel counts, unless the data is transmitted elsewhere.

Finally, the green LED is switched off, indicating both data acquisition and transmission have completed, and the program enters a final infinite loop.

I/O Pins

This example code was developed on an STMicroelectronics NUCLEO-U5A5ZJ-Q development board. The I/O pins were chosen to be easily accessible when using board. Any changes to I/O pin assignments should be made through the *rhd_acquisition.ioc* file.

SPI Bus (SPI3)

NSS ($\overline{CS}$): PA4

SCK (SCLK): PC10

MISO: PC11

MOSI: PC12

LEDs

LED_GREEN: PC7

LED_RED: PG2

Monitor Pins

Interrupt_Monitor: PD9 (This pin is written high each sample period, so its frequency represents amplifier sample rate.)

Main_Monitor: PC8 (This pin is written high as long as main loop is processing, so its duty cycle approximates free CPU clock cycles that could be used for other processing tasks.)

ErrorCode_Bit_3: PE0

ErrorCode_Bit_2: PG8

ErrorCode_Bit_1: PG5

ErrorCode_Bit_0: PG6

Connecting an Intan Chip

This example code was designed to work with an Intan RHD2132 32-channel amplifier chip. The RHD2216 chip can also be used, but unless the code is modified, half of the CONVERT commands per sample period will not correspond to real channels. The RHD2164 chip uses a non-standard double-data-rate SPI bus protocol, so connecting this chip to an STM32 would require some external "glue logic" and significant changes to this firmware which we have not yet implemented. The RHS2116 chip uses 32-bit instead of 16-bit SPI words, so it also would require significant firmware changes which are planned for later release.

Intan headstages are set to communicate using LVDS (low voltage differential signaling) signals on the SPI bus, while most microcontrollers use standard non-LVDS SPI signals. To connect an Intan RHD headstage to the STM32, you must either use an Intan **LVDS adapter board** (part #C3490), or tie the **LVDS_en** pin on the RHD chip to ground to disable LVDS signaling. On most Intan RHD headstages the LVDS_en pin is hard-wired to VDD and it is impractical to cut the trace, but in the C3335 headstage (which uses a RHD2132 chip with access to 16 of the 32 amplifiers), there is a zero-ohm resistor labeled R4 that can be removed to set LVDS_en low.

Note that if LVDS signaling is disabled, standard CMOS signaling is used instead, which makes reliable transmission of high-frequency data over long wires challenging due to signal reflections. For this reason, if LVDS is disabled, it is critical to keep the SPI wires between the MCU and the RHD chip as short as possible, and/or reduce the SPI data transmission rate.

Another helpful accessory for development is an RHD SPI cable adapter (part #C3430), which plugs into the SPI connector on an Intan headstage and breaks out each signal to a circuit board for easy soldering. Each of the twelve signals is assigned its own hole for soldering, from B1 - B6 (bottom row) and T1 - T6 (top row). (The SPI cable adapter is not necessary if the LVDS adapter board is used, because the LVDS adapter board includes an SPI cable connector.)

Assuming LVDS signaling is disabled and SPI signals are kept short, the following connections can be used along with the example program to interface an STM32U5 with an RHD2132 headstage through an SPI cable adapter board. Note that with LVDS disabled, all the (–) polarity signals are unused, and the (+) polarity signals carry the standard CMOS signal.

RHD SPI adapter board pin number	Signal	STM32 pin number
B1	$\overline{CS}$+	PA4
B2	SCLK+	PC10
B3	MOSI+	PC12
B4	MISO1+	PC11
B5	MISO2+ (unused)	-
B6	VDD	3V3
T1	$\overline{CS}$– (unused)	-
T2	SCLK– (unused)	-
T3	MOSI– (unused)	-
T4	MISO1– (unused)	-
T5	MISO2– (unused)	-
T6	GND	GND

Description of User-Changeable Sections of Example Code

The example code was designed with user modifications in mind. While users are free to modify any and all files, there are three specific files that are intended to be modified to most effectively alter the program's functionality: **userconfig.h**, **userfunctions.h**, and **userfunctions.c**. Each of these files and the changes that may be made to them are discussed in detail below. Note that the .ioc file, which governs pin and peripheral configuration, is not included here and should instead be changed directly through STM32CubeIDE's UI if the user wants to use different I/O pins, different peripherals, or different parameters for those peripherals (e.g., to change SPI baud rate or timer-generated sample rate).

userconfig.h

The user changes in this file are parameters set with `#define` macros. These parameters are used throughout various files of the project, but most aspects of the program that users will want to alter can simply be set here. For example, the number of channels converted per sequence, how many samples to store in memory, and which channel should have its samples stored. Those that require more involved changes (for example, sample rate) get further explanation in their sections.

```
#define USE_HAL
```

If this line is left uncommented, the code is compiled for compatibility with HAL (Hardware Abstraction Layer) drivers for all peripherals. It is important that if this is left uncommented, the user navigates to the STM32CubeIDE IOC viewer -> Project Manager -> Advanced Settings, and confirms that all the peripherals that are directly used in user code (GPIO, GPDMA, USART, TIM, SPI) are set to HAL. In contrast, if this line is commented out, the code is compiled for compatibility with LL (Low Layer) drivers, and these peripherals should all be set to LL instead.

In general, HAL drivers are more user-friendly, simple to work with, and largely compatible even across different STM32 series. LL drivers require more device-specific implementation of basic functions, and are closer to direct manipulation of registers, so the code using LL tends to be more complex but specialized to the chip, typically boosting performance.

```
#define ERROR_DETECTED_PORT          LED_RED_GPIO_Port
#define ERROR_DETECTED_PIN           LED_RED_Pin
```

These two lines specify the port and pin address of the "Error Detection" pin. By default, this is set to the pin that routes to a red LED on the Nucleo board. If another pin is desired to be used instead, that pin's Port and Pin addresses should be changed here.

```
#define CONVERT_COMMANDS_PER_SEQUENCE      32
```

This line specifies that for every sample period, 32 CONVERT commands will be sent within a single sequence. The order of these CONVERT commands can be customized in *configure_convert_commands()* in *userfunctions.c*, but if left unchanged, this will be channels 0-31 in ascending order.

```
#define AUX_COMMANDS_PER_SEQUENCE          3
```

This line specifies that for every sample period, after the CONVERT commands, three auxiliary commands will be sent within a single sequence. The contents of these auxiliary commands can be customized in *configure_aux_commands()* in *userfunctions.c*, but if left unchanged, this will have command 1 cycle through all RHD registers, re-writing each according to software-configured values, and commands 2 and 3 simply repeat dummy READ commands on ROM registers.

The total number of commands sent in a sequence of a single sample period is CONVERT_COMMANDS_PER_SEQUENCE + AUX_COMMANDS_PER_SEQUENCE. By default, this is 32 + 3 = 35, so at every period of 1 / SAMPLE_RATE, there will be a sequence of 35 individual 16-bit SPI command words.

```
#define AUX_COMMAND_LIST_LENGTH      128
```

This line specifies how many auxiliary commands are contained in a single auxiliary command list. Each of the AUX_COMMANDS_PER_SEQUENCE (by default three) has its own slot, and every sequence iteration (which happens once per sample period), all slots advance by one through their lists. Once the end of the list length (dictated by this parameter) is reached, all lists are reset to zero, and the next sequence all slots will

begin at the beginning of their list. In the default example, slot 0 reprograms most of the RHD RAM registers. Because all lists are 128 commands long, when a specific write occurs it can be expected that exactly 128 samples later, that write will be repeated during the next cycle of the aux command list.

Note that all auxiliary commands will have this same length, with the exception of Zcheck_DAC command lists, which have variable lengths because different DAC output signal frequencies will require different numbers of commands. Progress through these command lists is tracked separately, and will loop back to the start at some variable rate depending on frequency. All other command lists will loop back to their start every AUX_COMMAND_LIST_LENGTH number of samples.

```
#define SAMPLE_RATE             20000
```

This line specifies the sample rate in Hz of all amplifier channels on the chip. This parameter is used when determining suitable RHD register values for initial configuration, as some register fields like ADC buffer bias and MUX bias have optimum values that vary with sample rate. **Note**: Changing this value here alone does not change the actual sample rate of the example program. The sample period is governed by the TIM3 peripheral, specifically the clock input it receives and the counter period set in the .ioc file. The system clock speed of 160 MHz is divided by the counter period of 8000, resulting in the timer issuing an interrupt at the desired 20 kHz. **If a different sample rate is desired, it must be changed through the .ioc file by altering the TIM3 counter period**, not just setting this `#define` here.

```
#define SAMPLES_IN_MEMORY       20000
```

This line specifies how many samples are stored in memory before the main data acquisition loop is escaped. By default, once 20000 samples from a single channel have been acquired, the main loop exits, and acquisition stops. Those 20000 samples will then be transmitted off-chip via USART before the program terminates. With the default SAMPLE_RATE of 20000, this corresponds to one second of data acquisition. On-chip RAM is limited, so setting this number excessively high will cause the chip to run out of memory during program execution.

```
#define SELECTED_CHANNEL        8
```

This line specifies which of the 32 RHD amplifier channels is selected to have each sample stored in memory and ultimately transmitted via USART. The choice of amplifier 8 is arbitrary; any of the channels 0-31 can be used. While all 32 channels are sampled, with each receiving its own CONVERT command once per sample period, only this single specified channel has its samples written to memory. If multiple channels are instead desired, the user should alter the *write_data_to_memory()* function within *userfunctions.c*.

userfunctions.h

This header file contains the declarations of all functions in *userfunctions.c*, but also a few static inline functions that are short and simple enough to go directly in this .h file. Each of these functions is discussed below.

```
static inline void enable_interrupt_timer(int enable)
```

This function is called directly before the beginning of the main acquisition loop, with enable = 1, and directly after the end of the loop, with enable = 0. The function either starts or stops the timer and its ability to issue an interrupt when the timer reaches its target counter value. The exact implementation differs based on HAL and LL drivers, but generally the same behavior is achieved.

Users are not likely to change this function unless drastically changing how timers are used to generate interrupts.

```
static inline void send_convert_commands()
```

This function sends all CONVERT commands present in *command_sequence_MOSI* one-by-one via individual SPI transfers. Note that these SPI transfers are not optimized for speed (using DMA or interrupts), but just poll until they're complete. These are not efficient for high-performance transfers (see the our use of DMA to achieve these transfers more efficiently) but provide the most straightforward, basic way of sending polling SPI transfers. This function is not actually used in this example, as instead CONVERT commands are transferred via DMA, but it can be used with the knowledge that performance and efficiency will be reduced.

Users are not likely to change this function, but may find its simple implementation useful for understanding how CONVERT commands are sent via SPI.

`static inline void send_aux_commands()`

This function sends all auxiliary commands present in *aux_command_list* one-by-one via individual SPI transfers. As above, these SPI transfers are not optimized for speed, and as such this function is not actually used in the example.

Users are not likely to change this function, but may find its simple implementation useful for understanding how auxiliary commands are sent via SPI.

userfunctions.c

This file contains the implementations of various functions that are likely to be changed by the user to affect the behavior of the example program. Each of these functions is discussed below.

`int loop_escape()`

This function determines under what conditions the main acquisition loop is exited, and is called both within the main loop and within the interrupt routine to make sure that this condition's fulfilment is detected immediately. When this function returns 1, the main loop exits. If the user wishes to permanently stay in the main loop (for example, extended real time acquisition and transfer), this function can be written to always return 0. By default, this function returns 1 once *sample_counter* exceeds SAMPLES_IN_MEMORY, indicating that this number of samples has been acquired (one second at 20 kHz, or 20000 samples).

Users may want to change this function to check some other condition, or hard-code it to return 0 if the user never wants the main acquisition loop to be broken.

`void write_data_to_memory()`

This function writes the SELECTED_CHANNEL's most recent CONVERT command result from the *command_sequence_MISO* array into the *sample_memory* array, incrementing the *sample_counter* variable. The +2 offset used to index this MISO array is the result of the 2-command pipeline delay explained in the RHD chip datasheet: each MOSI command will see its MISO result 2 commands later, and so this function checks two results further down the pipeline than the original SELECTED_CHANNEL. Once the main data acquisition loop is escaped, the data in *sample_memory* is transmitted at once (offline, as opposed to real time) via USART.

Users may want to change this function for a variety of reasons:

If extended real time data acquisition is desired (if *loop_escape()* is intended to never return 1), this function should be left completely empty to avoid *sample_counter* from continuing to increment to very high levels and *sample_memory* eventually getting written out of bounds.

If data from multiple channels is desired to be saved, then multiple arrays like *sample_memory* can be used (or a single 2-D array serving the same purpose), so that every sample period a sample from each selected channel should be saved to memory somewhere. Note that on-chip RAM is finite, and saving too many samples from too many channels will eventually cause the STM32 chip to run out of memory.

Finally, if the auxiliary command slots are used for some more advanced purposes that require reading their results, they can be read here (as demonstrated in the commented-out code in the function). Due to the 2-command pipeline, and the fact that these are the last three commands in a command sequence, the result of AUX SLOT 1 will be from the current command sequence, whereas AUX SLOT 2 and AUX SLOT 3 will be from the previous command sequence.

`void transmit_data_realtime()`

This function transmits data from the selected channel, in real time (once per sample period) via USART. By default, this function is completely commented out, as the example instead demonstrates offline data transfer (waiting until the full acquisition period is finished, and then transferring all acquired data at once).

Users may want to change this function if real time acquisition is desired. The function *write_data_to_memory()* should be basically replaced by this function. Instead of reading a specific channel's sample and saving it to memory, transmit it via USART or whatever the desired peripheral is. Note that this executes once per sample period. If it takes too long, the next sample period will trigger before finishing. It is critical to avoid this important error condition, referred to as *ITClip*, so care must be taken to ensure this function does not take very long to complete. For example, if transmitting via USART, make sure data is sent quickly at a high baud rate.

`void transmit_data_offline()`

This function transmits acquired data from the selected channel, once the entire acquisition period has finished and the main loop escaped, via USART. Implementations using both HAL and LL drivers are included in this function, generally accomplishing the same behavior.

Users may want to change this function if they have made some change to the way that data is saved (for example, using multiple channels), or remove it completely for real time data. This function is only called after the main acquisition loop is broken, so for real time acquisitions in which the main loop never breaks, this function has no purpose, and most likely users will not want to send data both in real time and offline.

`void configure_registers()`

This function sets reasonable values for the RHD registers in the *RHDConfigParameters* struct, and writes them via SPI. These values are determined programmatically through the functions *write_initial_reg_values* and *set_default_rhd_settings*, in the *rhdinterface.c* and *rhdregisters.c* files, before being written via SPI.

Users who want to customize the values of specific registers before acquisition will want to change this function by altering *parameters* after *write_initial_reg_values* is called, and then writing a command specifically for each changed registers. A commented-out example is included in this function, demonstrating how registers 5 and 7 can be set to allow for an impedance measurement to occur. (Register 6 should be changed sample-to-sample via an aux command list.)

`void configure_convert_commands()`

This function sets up the CONVERT_COMMANDS_PER_SEQUENCE (default 32) CONVERT commands into the *command_sequence_MOSI* array, which is used every sample period to send all 32 commands in a single sequence. This implementation should call *create_convert_sequence* to populate each of these 32 commands as a 16-bit SPI word that the RHD chip will recognize. Passing NULL as the second parameter will automatically order these CONVERT commands from 0-31, otherwise an array of `uint8_t` numbers can be passed to specify a specific order for these commands to occur.

If CONVERT_COMMANDS_PER_SEQUENCE has been reduced for performance reasons, this array will specify which channels are sampled at all.

Users who want to alter the order of CONVERT commands or specifically leave out certain channels from conversions will want to change this function by altering the 2nd argument to *create_convert_sequence*. A commented-out example is included in this function, demonstrating how to create and pass a *channel_numbers* array so that the sequence populating *command_sequence_MOSI* is ordered from 31-0 instead of 0-31.

`void configure_aux_commands()`

This function sets up the AUX_COMMANDS_PER_SEQUENCE (default 3) auxiliary command lists that are loaded into the end of the *command_sequence_MOSI* array, which is used every sample period to send three auxiliary commands at the end of a single sequence. This implementation should call some variation of a *create_command_list* function for each of the (default 3) auxiliary command slots.

Users who want to alter the number of auxiliary commands, or the contents of each command list, will want to change this function by changing which *create_command_list* functions are called for each command slot. By default, slot 1 is an RHD register configuration command list, and slots 2 and 3 are dummy reads of ROM registers 40 and 41 respectively. Since impedance check command lists have variable lengths (all other command lists are created to be AUX_COMMAND_LIST_LENGTH commands long, by default 128), the

process for creating an impedance check command list also requires setting the variable *zcheck_DAC_command_slot_position*, and is demonstrated in the commented-out section of this function.

Performance Considerations

This STM32 example code is designed to run comfortably with either HAL or LL drivers, achieving a sample rate of 20 kS/s for 32 channels + 3 auxiliary commands per sample period, with most of the time during acquisition spent processing in the main loop: most CPU time is free for other processing tasks. However, some applications might need further performance improvements, for example if 30 kS/s is desired, or if multiple RHD chips are controlled with a single MCU. In these cases, there are some steps that can be taken to optimize performance for specific applications.

HAL vs LL

HAL (Hardware Abstraction Layer) tends to have more simple function calls and is more uniform across all STM32 chip series, sacrificing efficiency for simplicity. LL (Low Layer) allows for more efficient completion of given tasks, but requires a deeper understanding of the individual STM32 registers. We recommend users start with HAL to gain a general understanding of how the example program works, and then if more advanced understanding is required switch to LL to see how the general behavior achieved by HAL can be implemented closer to the register level. LL functions tend to also be executed much faster than HAL functions, so if the user reaches a performance bottleneck often simply switching from HAL to LL will speed up execution dramatically.

Increasing sample rate

While changing the `#define SAMPLE_RATE` value and the counter period option for TIM3 in the .ioc will change the sample rate, rates beyond 20 kS/s are very likely to cause each sample period interrupt to clip into the next (*ITClip* error), halting program execution and illuminating the red LED. Each action of the interrupt routine will contribute to this, but the most likely culprit is the SPI sequence taking too long to complete. The SPI sequence's speed is physically limited by the minimum time requirements outlined in the RHD datasheet, and the example program is already quite close to these minimum requirements. The details of streamlining the SPI transfer are discussed below.

However, the simplest way to get a 35-command sequence to finish executing before the next sample rate is to reduce the number of commands below 35. Since the total length of the command sequence is CONVERT_COMMANDS_PER_SEQUENCE + AUX_COMMANDS_PER_SEQUENCE, reducing either of these will reduce the amount of time required for the sequence to complete. If fewer than 32 channels are required, CONVERT_COMMANDS_PER_SEQUENCE can be reduced to only include the channels that are sampled. (In the default example program, only one of the 32 sampled channels actually has its data saved.) Similarly, of the three auxiliary command slots that are included in the example program, two are dummy command lists that only read ROM registers and act as placeholders for any other auxiliary command lists, so many users will find AUX_COMMANDS_PER_SEQUENCE can be reduced. The first command slot continually reprograms the RHD registers, which allows for quick recovery from any unexpected data corruption during acquisition, and is a good idea for longer acquisition sessions but is not strictly necessary if the speed-up from removing a single SPI command word is critical.

Streamlining SPI Communication

In addition to reducing the number of SPI words per sample period, some steps can be taken to make each SPI word faster. Currently, the SPI achieves a 20 Mbit/s baud rate (the speed at which SCLK switches throughout a 16-bit word) by dividing the 160 MHz clock by a 8x prescaler. The maximum SCLK baud rate the RHD chip accepts is 25 MHz, as seen on the RHD chip datasheet. So, there can be a significant SPI speed increase by setting the baud rate to 25 Mbit/s – however, this would require a clock speed of 100 MHz (4x prescaler) or 200 MHz (8x prescaler). The STM32U5 chip is capable of a maximum clock speed of 160 MHz, so the SPI communication can actually be sped up by reducing the clock speed to 100 MHz and using a 4x instead of 8x prescaler. Obviously, throttling the clock speed will reduce efficiency of all other processing tasks, but the example program does not require very intensive processing so users may find this significantly boosts SPI performance while not incurring too much slow down elsewhere.

A significant source of slow-down is that the SPI peripheral seems to require at least ~40 ns between $\overline{CS}$ driving low and the first SCLK pulse. While we have not found a way to speed this up, there may be some capability we are unaware of that allows for this latency to be reduced closer to the minimum 20 ns listed on the RHD datasheet. Perhaps other STM32 chip series with more efficient SPI implementations or higher clock speeds are capable of initiating the SPI transfer sooner after $\overline{CS}$ drives low.

Another important note when discussing changing SPI timing is ensuring the minimum time requirements from the RHD datasheet are met. If SPI speed is changed, the user must make sure these parameters (namely maximum SCLK frequency of 25 MHz and $\overline{CS}$ stays high for 154 ns between words) stay valid. The example program already uses MIDI (Master Inter-Data Idleness), and other SPI control fields like MSSI (Master SS Idleness) in the SPI_CFG2 register can be used to ensure the timing requirements are not violated.

Speeding up USART

The example code transmits data offline, waiting until a full second's worth of data is acquired before transmitting any data via USART, but this can be altered to transmit data in real time instead. See *transmit_data_realtime()* for a discussion on how to do this. For offline data transfer speed is unimportant, but if data needs to be transferred alongside data acquisition, the USART transmission must be as fast as possible to avoid impacting data sampling. While the details of USART communication are beyond the scope of this document, the higher the baud rate the more quickly data can be sent. As long as the targeted USART reader is capable of receiving data at a high baud rate, the user should consider increasing the baud rate. The example code uses 5 Mbit/s which is more than sufficient for streaming a single channel's data in real time, and is likely capable of increasing further to at least 10 Mbit/s.

Future Plans

This firmware framework has been developed to use a single SPI bus with an STM32U5 MCU to communicate with a single RHD2216 or RHD2132. However, we plan to expand this firmware framework to ultimately include a variety of other features, including the following. If there are other features suitable for your application that you'd like to see in future STM32 framework releases, please provide us your feedback at **support@intantech.com**.

1. Multiple SPI buses on a single MCU to control multiple Intan chips at once. Assuming efficient SPI communication can be achieved using features like DMA, the increased processing load for multiple SPI buses should remain minimal, so that adding more SPI instances without slowing down sample rate is *probably* easy. This assumption will not hold for MCUs with SPI implementations similar to the F4, which require direct processor intervention between each SPI word to pulse NSS high. However, more recent STM32 chips with more advanced SPI implementations that do not require this direct processor intervention should all be capable of very efficient SPI communication, so we expect them to be able to handle multiple SPI buses simultaneously.

2. Various STM32 series other than U5 (for example H7, WL, and/or WBA series), depending on MCU capabilities and popular demand. While the big-picture control of SPI and DMA peripherals to communicate with Intan chips will remain the same across various series, each series may have significant changes to the implementations of their peripherals. We expect that the HAL implementation for different series will generally be quite similar, but the LL implementations that are closer to the register level may vary significantly.

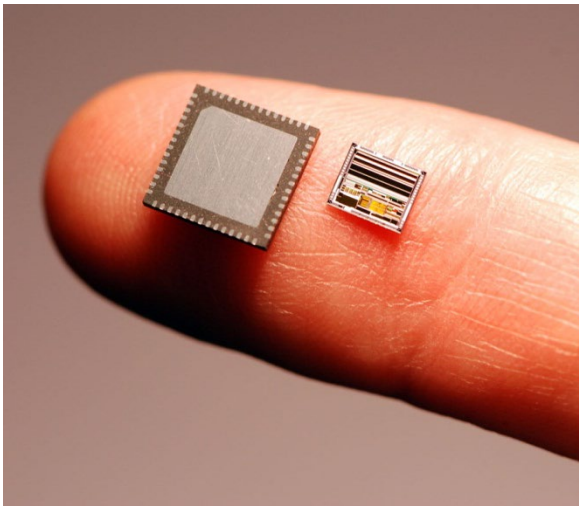
3. SPI control of RHS2116 chips. The RHS SPI communication protocol differs significantly from the standard RHD iteration: each SPI word is 32 bits instead of 16 bits. Another important element is the requirement to supply VSTIM+ and VSTIM-, which are usually set to +7 V and -7 V respectively, and these voltages are not typically accessible from an MCU directly.

4. SPI control of RHD2164 chips. The RHD2164 SPI communication protocol differs significantly from other Intan chips in that it uses non-standard double-data-rate (DDR) SPI data transfer, in which MISO is sampled at both the rising and falling edges of SCLK. This should be achievable by configuring the STM32 SPI peripheral to double baud rate with 32-bit words (instead of 16-bit words) and adding a small amount of additional external hardware to divide the SCLK frequency by two.

Contact Information

This datasheet is meant to acquaint engineers and scientists with the Intan STM32 interface code developed at Intan Technologies. We value feedback from potential end users. We can discuss your specific needs and suggest a solution for your applications.

For more information, contact Intan Technologies at:



intan TECHNOLOGIES, LLC

www.intantech.com
support@intantech.com